Rails 风格指南

wizardforcel

Published with GitBook

目錄

介绍 0
Rails 风格指南 1
配置 2
路由 3
控制器 4
模型 5
迁移 6
视图 7
国际化 8
Assets 9
Mailers 10
Time 11
Bundler 12
有缺陷的 Gem 13
进程管理 14
延伸阅读 15
贡献 16
许可证 17
口耳相传 18

序幕

> 榜样很重要。
> -- 《机械战警》 Alex J. Murphy 警官

这份指南旨在提供一系列 Ruby on Rails 4 开发的最佳实践和风格惯例。本指南与社区驱动并制定的 Ruby 编码风格指南可以互为补充。

本文中的一些建议只适用于 Rails 4.0+ 版本。

你可以使用 Transmuter 来生成本文的 PDF 或 HTML 版本。

本指南同时有以下语言的翻译版：

- 英文原版
- 繁體中文
- 日语
- 俄语
- 土耳其语

Rails 风格指南

这份 Rails 风格指南推荐的是 Rails 的最佳实践，现实世界中的 Rails 程序员据此可以写出可维护的高质量代码。我们只说实际使用中的用法。指南再好，但里面说的过于理想化结果大家拒绝使用或者可能根本没人用，又有何意义。

本指南分为几个小节，每一小节由几条相关的规则构成。我尽力在每条规则后面说明理由（如果省略了说明，那是因为其理由显而易见）。

这些规则不是我凭空想象出来的——它们中的绝大部分来自我多年以来作为职业软件工程师的经验，来自 Rails 社区成员的反馈和建议，以及许多备受推崇的 Rails 编程资源。

目录

- 配置
- 路由
- 控制器
- 模型
    - ActiveRecord
    - ActiveRecord 查询
- 迁移
- 视图
- 国际化
- Assets
- Mailers
- Time
- Bundler
- 有缺陷的 Gem
- 进程管理

配置

- 自定义的初始化代码应放在 `config/initializers` 目录下。 Initializers 目录中的代码在应用启动时被执行。 [link]
- 每个 gem 的初始化代码应放在单独的文件中，并且文件名应与 gem 的名称相同。例如： `carrierwave.rb` , `active_admin.rb` 。 [link]
- 相应地调整开发环境、测试环境及生产环境的配置（修改 `config/environments/` 目录下对应的文件） [link]
    - 添加需要预编译的额外静态资源文件（如果有的话）：

      ```
      ```Ruby
      # config/environments/production.rb
      # 预编译额外的静态资源文件(application.js, application.css, 以及
      config.assets.precompile += %w( rails_admin/rails_admin.css
      ```
      ```

- 将所有环境下都通用的配置放在 `config/application.rb` 文件中。 [link]
- 创建一个与生产环境高度相似的 `staging` 环境。 [link]
- 其它配置应保存在 YAML 文件中，存放在 `config/` 目录下。 [link]

  从 Rails 4.2 开始，可以通过 `config_for` 这个新方法轻松地加载 YAML 配置文件：

  ```
  Rails::Application.config_for(:yaml_file)
  ```

路由

- 当需要为一个 RESTful 资源添加动作时（你真的需要吗？），应使用 `member` 路由和 `collection` 路由。[link]

```
# 差
get 'subscriptions/:id/unsubscribe'
resources :subscriptions

# 好
resources :subscriptions do
  get 'unsubscribe', on: :member
end

# 差
get 'photos/search'
resources :photos

# 好
resources :photos do
  get 'search', on: :collection
end
```

- 当需要定义多个 `member/collection` 路由时，应使用块结构。[link]

```
resources :subscriptions do
  member do
    get 'unsubscribe'
    # 更多路由
  end
end

resources :photos do
  collection do
    get 'search'
    # 更多路由
  end
end
```

- 使用嵌套路由(nested routes)，它可以更有效地表现 ActiveRecord 模型之间的关系。[link]

```ruby
class Post < ActiveRecord::Base
  has_many :comments
end

class Comments < ActiveRecord::Base
  belongs_to :post
end

# routes.rb
resources :posts do
  resources :comments
end
```

- 使用命名空间路由来归类相关的动作。[link]

```ruby
namespace :admin do
  # 将请求 /admin/products/* 交由 Admin::ProductsController 处理
  # (app/controllers/admin/products_controller.rb)
  resources :products
end
```

- 不要使用旧式的控制器路由。这种路由会让控制器的所有动作都通过 GET 请求调用。[link]

```ruby
# 非常差
match ':controller(/:action(/:id(.:format)))'
```

- 不要使用 `match` 来定义任何路由，除非确实需要将多种请求映射到某个动作，这时可以通过 `via` 选项来指定请求类型，如 `[:get, :post, :patch, :put, :delete]` 。[link]

控制器

- 控制器应该保持苗条 — 它们应该只为视图层提供数据，不应包含任何业务逻辑（所有业务逻辑都应当放在模型里）。[link]
- 每个控制器的动作（理论上）应当只调用一个除了初始的 find 或 new 之外的方法。[link]
- 控制器与视图之间共享不超过两个实例变量。[link]

模型

- 自由地引入不是 ActiveRecord 的模型类。[link]
- 模型的命名应有意义（但简短）且不含缩写。[link]
- 如果需要模型类有与 ActiveRecord 类似的行为（如验证），但又不想有 ActiveRecord 的数据库功能，应使用 ActiveAttr 这个 gem。[link]

```
class Message
  include ActiveAttr::Model

  attribute :name
  attribute :email
  attribute :content
  attribute :priority

  attr_accessible :name, :email, :content

  validates_presence_of :name
  validates_format_of :email, :with => /\A[-a-z0-9_+\.]+\@([-a-
  validates_length_of :content, :maximum => 500
end
```

更完整的示例请参考 RailsCast on the subject。

ActiveRecord

- 避免改动缺省的 ActiveRecord 惯例（表的名字、主键等），除非你有一个充分的理由（比如，不受你控制的数据库）。[link]

```
# 差 - 如果你能更改数据库的 schema，那就不要这样写
class Transaction < ActiveRecord::Base
  self.table_name = 'order'
  ...
end
```

- 把宏风格的方法调用（`has_many`, `validates` 等）放在类定义语句的最前面。[link]

```ruby
class User < ActiveRecord::Base
  # 默认的 scope 放在最前（如果有的话）
  default_scope { where(active: true) }

  # 接下来是常量初始化
  COLORS = %w(red green blue)

  # 然后是 attr 相关的宏
  attr_accessor :formatted_date_of_birth

  attr_accessible :login, :first_name, :last_name, :email, :pas

  # 紧接着是与关联有关的宏
  belongs_to :country

  has_many :authentications, dependent: :destroy

  # 以及与验证有关的宏
  validates :email, presence: true
  validates :username, presence: true
  validates :username, uniqueness: { case_sensitive: false }
  validates :username, format: { with: /\A[A-Za-z][A-Za-z0-9._-
  validates :password, format: { with: /\A\S{8,128}\z/, allow_r

  # 下面是回调方法
  before_save :cook
  before_save :update_username_lower

  # 其它的宏（如 devise）应放在回调方法之后

  ...
end
```

- `has_many :through` 优于 `has_and_belongs_to_many`。使用 `has_many :through` 允许 join 模型有附加的属性及验证。[link]

```ruby
# 不太好 - 使用 has_and_belongs_to_many
class User < ActiveRecord::Base
  has_and_belongs_to_many :groups
end

class Group < ActiveRecord::Base
  has_and_belongs_to_many :users
end

# 更好 - 使用 has_many :through
class User < ActiveRecord::Base
  has_many :memberships
  has_many :groups, through: :memberships
end

class Membership < ActiveRecord::Base
  belongs_to :user
  belongs_to :group
end

class Group < ActiveRecord::Base
  has_many :memberships
  has_many :users, through: :memberships
end
```

- `self[:attribute]` 比 `read_attribute(:attribute)` 更好。[link]

```ruby
# 差
def amount
  read_attribute(:amount) * 100
end

# 好
def amount
  self[:amount] * 100
end
```

- `self[:attribute] = value` 优于 `write_attribute(:attribute, value)` 。[link]

```
# 差
def amount
  write_attribute(:amount, 100)
end

# 好
def amount
  self[:amount] = 100
end
```

- 总是使用新式的 "sexy" 验证。[link]

```
# 差
validates_presence_of :email
validates_length_of :email, maximum: 100

# 好
validates :email, presence: true, length: { maximum: 100 }
```

- 当一个自定义的验证规则使用次数超过一次时，或该验证规则是基于正则表达式时，应该创建一个自定义的验证规则文件。[link]

```
# 差
class Person
  validates :email, format: { with: /\A([^@\s]+)@((?:[-a-z0-9]+
end

# 好
class EmailValidator < ActiveModel::EachValidator
  def validate_each(record, attribute, value)
    record.errors[attribute] << (options[:message] || 'is not a
  end
end

class Person
  validates :email, email: true
end
```

- 自定义验证规则应放在 `app/validators` 目录下。[link]
- 如果你在维护数个相关的应用，或验证规则本身足够通用，可以考虑将自定义的验证规则抽象为一个共用的 gem。[link]
- 自由地使用命名 scope。[link]

```
class User < ActiveRecord::Base
  scope :active, -> { where(active: true) }
  scope :inactive, -> { where(active: false) }

  scope :with_orders, -> { joins(:orders).select('distinct(user
end
```

- 当一个由 lambda 和参数定义的命名 scope 太过复杂时， 更好的方式是创建一个具有同样用途并返回 `ActiveRecord::Relation` 对象的类方法。这很可能让 scope 更加精简。 [link]

```
class User < ActiveRecord::Base
  def self.with_orders
    joins(:orders).select('distinct(users.id)')
  end
end
```

注意这种方式不允许命名 socpe 那样的链式调用。例如：

```
# 不能链式调用
class User < ActiveRecord::Base
  def User.old
    where('age > ?', 80)
  end

  def User.heavy
    where('weight > ?', 200)
  end
end
```

这种方式下 `old` 和 `heavy` 可以单独工作，但不能执行 `User.old.heavy` 。 若要链式调用，请使用下面的代码：

```
# 可以链式调用
class User < ActiveRecord::Base
  scope :old, -> { where('age > 60') }
  scope :heavy, -> { where('weight > 200') }
end
```

- 注意 `update_attribute` 方法的行为。它不运行模型验证（与 `update_attributes` 不同），因此可能弄乱模型的状态。 [link]

- 应使用对用户友好的 URL。URL 中应显示模型的一些具有描述性的属性，而不是仅仅显示 `id` 。有多种方法可以达到这个目的：[link]

  - 重写模型的 `to_param` 方法。Rails 使用该方法为对象创建 URL。该方法默认会以字符串形式返回记录的 `id` 项。 可以重写该方法以包含其它可读性强的属性。

    ```
    class Person
      def to_param
        "#{id} #{name}".parameterize
      end
    end
    ```

  为了将结果转换为一个 URL 友好的值，字符串应该调用 `parameterize` 方法。 对象的 `id` 属性值需要位于 URL 的开头，以便使用 ActiveRecord 的 `find` 方法查找对象。

  - 使用 `friendly_id` 这个 gem。它允许使用对象的一些描述性属性而非 `id` 来创建可读性强的 URL。

    ```
    class Person
      extend FriendlyId
      friendly_id :name, use: :slugged
    end
    ```

  查看 gem documentation 以获得更多 `friendly_id` 的使用信息。

- 应使用 `find_each` 来迭代一系列 ActiveRecord 对象。用循环来处理数据库中的记录集（如 `all` 方法）是非常低效率的，因为循环试图一次性得到所有对象。而批处理方法允许一批批地处理记录，因此需要占用的内存大幅减少。[link]

```ruby
# 差
Person.all.each do |person|
  person.do_awesome_stuff
end

Person.where('age > 21').each do |person|
  person.party_all_night!
end

# 好
Person.find_each do |person|
  person.do_awesome_stuff
end

Person.where('age > 21').find_each do |person|
  person.party_all_night!
end
```

- 因为 Rails 为有依赖关系的关联添加了回调方法，应总是调用 `before_destroy` 回调方法，调用该方法并启用 `prepend: true` 选项会执行验证。[link]

```ruby
# 差——即使 super_admin 返回 true，roles 也会自动删除
has_many :roles, dependent: :destroy

before_destroy :ensure_deletable

def ensure_deletable
  fail "Cannot delete super admin." if super_admin?
end

# 好
has_many :roles, dependent: :destroy

before_destroy :ensure_deletable, prepend: true

def ensure_deletable
  fail "Cannot delete super admin." if super_admin?
end
```

ActiveRecord 查询

- 不要在查询中使用字符串插值，它会使你的代码有被 SQL 注入攻击的风险。[link]

```
# 差——插值的参数不会被转义
Client.where("orders_count = #{params[:orders]}")

# 好——参数会被适当转义
Client.where('orders_count = ?', params[:orders])
```

- 当查询中有超过 1 个占位符时，应考虑使用名称占位符，而非位置占位符。[link]

```
# 一般般
Client.where(
  'created_at >= ? AND created_at <= ?',
  params[:start_date], params[:end_date]
)

# 好
Client.where(
  'created_at >= :start_date AND created_at <= :end_date',
  start_date: params[:start_date], end_date: params[:end_date]
)
```

- 当只需要通过 id 查询单个记录时，优先使用 `find` 而不是 `where` 。[link]

```
# 差
User.where(id: id).take

# 好
User.find(id)
```

- 当只需要通过属性查询单个记录时，优先使用 `find_by` 而不是 `where` 。[link]

```
# 差
User.where(first_name: 'Bruce', last_name: 'Wayne').first

# 好
User.find_by(first_name: 'Bruce', last_name: 'Wayne')
```

- 当需要处理多条记录时，应使用 `find_each` 。[link]

```ruby
# 差——一次性加载所有记录
# 当 users 表有成千上万条记录时，非常低效
User.all.each do |user|
  NewsMailer.weekly(user).deliver_now
end

# 好——分批检索记录
User.find_each do |user|
  NewsMailer.weekly(user).deliver_now
end
```

- `where.not` 比书写 SQL 更好。[link]

```ruby
# 差
User.where("id != ?", id)

# 好
User.where.not(id: id)
```

迁移

- 应使用版本控制工具记录 `schema.rb` （或 `structure.sql` ）的变化。[link]

- 应使用 `rake db:scheme:load` 而不是 `rake db:migrate` 来初始化空数据库。[link]

- 应在迁移文件中设置默认值，而不是在应用层面设置。[link]

  ```ruby
  # 差——在应用中设置默认值
  def amount
    self[:amount] or 0
  end
  ```

  虽然许多 Rails 开发者建议在 Rails 中强制使用表的默认值，但这会使数据受到许多应用 bug 的影响，因而导致应用极其难以维护。考虑到大多数有一定规模的 Rails 应用都与其它应用共享数据库，保持应用的数据完整性几乎是不可能的。

- 务必使用外键约束。在 Rails 4.2 中，ActiveRecord 本身已经支持外键约束。[link]

- 书写建设性的迁移（添加表或列）时，应使用 `change` 方法而不是 `up` 或 `down` 方法。[link]

  ```ruby
  # 老式写法
  class AddNameToPeople < ActiveRecord::Migration
    def up
      add_column :people, :name, :string
    end

    def down
      remove_column :people, :name
    end
  end

  # 新式写法（更好）
  class AddNameToPeople < ActiveRecord::Migration
    def change
      add_column :people, :name, :string
    end
  end
  ```

- 不要在迁移中使用模型类。由于模型的变化，模型类也一直处在变化当中，过去运行正常的迁移可能不知什么时候就不能正常进行了。[link]

视图

- 不要直接从视图调用模型层。 [link]
- 复杂的格式化不应放在视图中，而应提取为视图 helper 或模型中的方法。 [link]
- 应使用 partial 模版与布局来减少代码重复。 [link]

国际化

- 不应在视图、模型或控制器里添加语言相关的设置，应在 `config/locales` 目录下进行设置。[link]

- 当 ActiveRecord 模型的标签需要被翻译时，应使用 `activerecord` scope: [link]

  ```
  en:
    activerecord:
      models:
        user: Member
      attributes:
        user:
          name: "Full name"
  ```

  然后 `User.model_name.human` 会返回 "Member" ，而 `User.human_attribute_name("name")` 会返回 "Full name"。这些属性的翻译会作为视图中的标签。

- 把在视图中使用的文字与 ActiveRecord 的属性翻译分开。把模型使用的语言文件放在 `models` 目录下，把视图使用的文字放在 `views` 目录下。[link]

  - 当使用额外目录来设置语言文件时，应在 `application.rb` 文件里列出这些目录以加载设置。

    ```ruby
    # config/application.rb
    config.i18n.load_path += Dir[Rails.root.join('config', 'l
    ```

- 把共享的本地化选项，如日期或货币格式，放在 `locales` 的根目录下。[link]

- 应使用精简形式的 I18n 方法：使用 `I18n.t` 而非 `I18n.translate`；使用 `I18n.l` 而非 `I18n.localize`。[link]

- 应使用 "懒惰" 查询来获取视图中使用的文本。假设我们有以下结构：[link]

  ```
  en:
    users:
      show:
        title: "User details page"
  ```

`users.show.title` 的数值能这样被 `app/views/users/show.html.haml` 获取：

```
= t '.title'
```

- 应在控制器与模型中使用点分隔的键，而非指定 `:scope` 选项。点分隔的调用更容易阅读，也更易追踪层级关系。[link]

```
# 差
I18n.t :record_invalid, :scope => [:activerecord, :errors, :mes

# 好
I18n.t 'activerecord.errors.messages.record_invalid'
```

- 更详细的 Rails i18n 信息可以在 Rails Guides 找到。[link]

Assets

应使用 assets pipeline 来管理应用的资源结构。

- 自定义的样式表、JavaScript 文件或图片文件，应放在 `app/assets` 目录下。[link]
- 把自己开发但不好归类的库文件，应放在 `lib/assets/` 目录下。[link]
- 第三方代码，如 jQuery 或 bootstrap，应放在 `vendor/assets` 目录下。[link]
- 尽可能使用资源的 gem 版。例如： jquery-rails, jquery-ui-rails, bootstrap-sass, zurb-foundation [link]

Mailers

- 应将 mailer 命名为 `SomethingMailer` 。若没有 Mailer 后缀，不能立即断定它是否为一个 mailer，也不能断定哪个视图与它有关。[link]

- 提供 HTML 与纯文本两份视图模版。[link]

- 在开发环境下应显示发信失败错误。这些错误默认是关闭的。[link]

```
# config/environments/development.rb

config.action_mailer.raise_delivery_errors = true
```

- 在开发环境下使用诸如 Mailcatcher 的本地 SMTP 服务器。[link]

```
# config/environments/development.rb

config.action_mailer.smtp_settings = {
  address: 'localhost',
  port: 1025,
  # 更多设置
}
```

- 为域名设置默认项。[link]

```
# config/environments/development.rb
config.action_mailer.default_url_options = { host: "#{local_ip}

# config/environments/production.rb
config.action_mailer.default_url_options = { host: 'your_site.c

# 在 mailer 类中
default_url_options[:host] = 'your_site.com'
```

- 若需要在邮件中添加到网站的超链接，应总是使用 `_url` 方法，而非 `_path` 方法。 `_url` 方法产生的超链接包含域名，而 `_path` 方法产生相对链接。[link]

```
# 差
You can always find more info about this course
<%= link_to 'here', course_path(@course) %>

# 好
You can always find more info about this course
<%= link_to 'here', course_url(@course) %>
```

- 正确地设置寄件人与收件人地址的格式。应使用下列格式：[link]

```
# 在你的 mailer 类中
default from: 'Your Name <info@your_site.com>'
```

- 确保测试环境下的 email 发送方法设置为 `test` ：[link]

```
# config/environments/test.rb

config.action_mailer.delivery_method = :test
```

- 开发环境和生产环境下的发送方法应设置为 `smtp` ：[link]

```
# config/environments/development.rb, config/environments/produ

config.action_mailer.delivery_method = :smtp
```

- 当发送 HTML 邮件时，所有样式应为行内样式，这是因为某些客户端不能正确显示外部样式。然而，这使得邮件难以维护理并会导致代码重复。有两个类似的 gem 可以转换样式，并将样式放在对应的 html 标签里： premailer-rails3 和 roadie。[link]
- 避免在产生页面响应的同时发送邮件。若有多个邮件需要发送，这会导致页面加载延迟甚至请求超时。有鉴于此，应使用 sidekiq 这个 gem 在后台发送邮件。[link]

Time

- 在 `application.rb` 里设置相应的时区。[link]

```
config.time_zone = 'Eastern European Time'
# 可选配置——注意取值只能是 :utc 或 :local 中的一个（默认为 :utc）
config.active_record.default_timezone = :local
```

- 不要使用 `Time.parse` 。[link]

```
# 差
Time.parse('2015-03-02 19:05:37') # => 会假设时间是基于操作系统的时

# 好
Time.zone.parse('2015-03-02 19:05:37') # => Mon, 02 Mar 2015 19
```

- 不要使用 `Time.now` 。[link]

```
# 差
Time.now # => 无视所配置的时区，返回操作系统时间。

# 好
Time.zone.now # => Fri, 12 Mar 2014 22:04:47 EET +02:00
Time.current # 结果同上，但更简洁
```

Bundler

- 只在开发环境或测试环境下使用的 gem 应进行适当的分组。[link]
- 在项目中只使用广为人知的 gem。如果你考虑引入某些鲜为人所知的 gem，应该先仔细检查一下其源代码。[link]
- 关于多个开发者使用不同操作系统的项目，与操作系统有关的 gem 默认情况下会产生经常变动的 `Gemfile.lock` 。在 Gemfile 文件里，所有与 OS X 相关的 gem 放在 `darwin` 群组，而所有与 Linux 有关的 gem 应放在 `linux` 群组：[link]

  ```ruby
  # Gemfile
  group :darwin do
    gem 'rb-fsevent'
    gem 'growl'
  end

  group :linux do
    gem 'rb-inotify'
  end
  ```

  要在正确的环境下加载合适的 gem，需添加以下代码至 `config/application.rb` ：

  ```ruby
  platform = RUBY_PLATFORM.match(/(linux|darwin)/)[0].to_sym
  Bundler.require(platform)
  ```

- 不要把 `Gemfile.lock` 文件从版本控制里移除。这可不是一个随机产生的文件——它的目的是确保你所有的团队成员执行 `bundle install` 时，获得相同版本的 gem 。[link]

有缺陷的 Gem

这是一个有问题的或有更好替代物的 gem 列表。不要在项目中使用它们。

- rmagick - 这个 gem 因大量消耗内存而臭名昭著。应使用 minimagick 来替代它。
- autotest - 测试自动化的过时方案，远不及 guard 和 watchr。
- rcov - 代码覆盖率工具，不兼容 Ruby 1.9。应使用 SimpleCov 来替代它。
- therubyracer - 内存杀手，强烈不建议在生产环境中使用。建议使用 `node.js` 来替代它。

这仍是一个完善中的列表，欢迎添加流行但有缺陷的 gem。

进程管理

- 如果项目依赖各种外界的进程，应使用 foreman 来管理它们。[link]

延伸阅读

以下是几个极好的讲述 Rails 风格的资源，闲暇时可以考虑延伸阅读：

- The Rails 4 Way
- Ruby on Rails Guides
- The RSpec Book
- The Cucumber Book
- Everyday Rails Testing with RSpec
- Better Specs for RSpec

贡献

本指南的每条建议都不是定案。我渴望与对 Rails 编码风格有兴趣的大家一起协作，创造出一份对整个 Ruby 社区都有益的资源。

欢迎 open tickets 或发送带有改进的 pull request。在此提前感谢您的帮助！

您可以通过 gittip 对本项目（以及 RuboCop 项目）进行捐赠支持。

如何贡献？

只需遵循贡献指南即可。

许可证

This work is licensed under a Creative Commons Attribution 3.0 Unported License

口耳相传

一份社区驱动的风格指南，若不为人所知，那有何用。请在微博转发这份指南，分享给你的朋友或同事。我们得到的每个评论、建议或意见都可以让这份指南变得更好一点。而我们想要拥有的是尽可能好的指南，不是吗？

共勉之，
Bozhidar